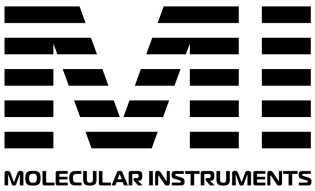


Molecular Instruments

Agentic AI Integration

Capstone Report

Department of Computer Science
University of California, Irvine

Instructors: Prof. Brian Vegetabile, Prof. Faisal Nawab

Mohamad Chaalan — 51802160 — mchaalan@uci.edu

Edom Eshete — 78341100 — eseshete@uci.edu

Kenny Gao — 90542960 — kennyg2@uci.edu

Anastassiya Iskaliyeva — 81764656 — aiskaliy@uci.edu

Angela Lombard — 74764556 — lombara1@uci.edu

Richard Wicaksono — 91968107 — wicaksor@uci.edu

Bitbucket Repository

https://bitbucket.org/molecularinstruments

December 11, 2025

1 Introduction and Problem Statement

Molecular Instruments (MI) is a biotechnology company that serves a highly technical customer base of primarily academic researchers and biopharma scientists. Across the company, teams rely on data and domain knowledge to make operational decisions, support customers and understand product usage. However, information is fragmented across internal documents, blog posts, protocol guides, research papers, scientific resources and across the SQL webstore database. Accessing this information typically requires some specialized skill such as being able to write complex SQL queries, or having the technical know how to answer very niche and domain-specific questions.

The goal of this capstone project is to design and implement a working **proof of concept (POC)** that utilizes AI agents to democratize Molecular Instruments data by making it more accessible for the internal staff to interact with these data sources, in turn, reducing bottlenecks created by manual search process or by waiting for the data team to create the needed SQL query. The **NL2SQL agent** enables non-technical employees to ask natural-language analytics questions and get back a valid executable SQL query which can be used in the companies analytics tool called **Metabase**. The **FAQ RAG agent** retrieves answers from MI's internal and public documents which helps eliminate staff from manually searching through the plethora of sources needed to answers customers niche and technical questions. The agents have been demonstrated to our sponsors and have received positive feedback but need to be further improved in order to be adopted. The NL2SQL agent is closer to the deployment phase and is performing as intended for its particular use case and audience. The FAQ RAG agent also shows potential but will still need refinement as we scale the corpus and continue rigorous testing. Future improvements will focus on better document chunking, prompting, and temperature parameter tuning as it gets scaled out and prepared for a broader roll out. These agents are not yet production-ready, but are functional prototypes that MI can evaluate for real-world usefulness, guide future development, and assess for potential integration into existing workflows.

2 Related Work

Agentic AI integration into business workflows is still relatively new; however, organizations exploring these workflows have introduced a range of techniques and tools that form the foundation for the solution addressed in this work. This includes prior research on context-aware AI agents for natural language–to–SQL translation as well as retrieval-augmented generation over internal company documents. The foundation of our Agents is based on **Google's Agent Development Kit (ADK)**. ADK is a flexible and modular framework for developing and deploying AI agents. An Agent is a self-contained execution unit designed to act autonomously to achieve specific goals powered by a **Large Language Model (LLM)**. Agents can perform tasks, interact with users, utilize external tools, and coordinate with other agents. The documentation [Goo24b] provided by Google is a rich resource for examples and best practices, which served as the main tool when developing and creating the **Molecular Instruments Agentic Workflow**.

There were two main works that inspired the NL2SQL Agent. The first one is from Per Jacobsson, a Principal Software Engineer for Google, who created a blog post entitled: *Getting AI to write good SQL: Text-to-SQL techniques explained* [Jac25], which helped shape the architecture design for our NL2SQL agent. The second one is a research report entitled *Large Language Model Enhanced Text-to-SQL Generation: A Survey* [ZLCL24] that compared the different techniques behind developing an NL to SQL tool. It discussed some of the common pitfalls and difficulties behind the task at hand, which helped us learn from the challenges others have faced. There are many works that discuss **Retrieval-Augmented Generation (RAG)**, but the one that offers the most practical insight is Google's paper entitled *Deeper insights into retrieval augmented generation: The role of sufficient context* [RJ25], which discusses the importance of providing sufficient external context to ground LLMs. This work highlights how properly retrieved context enables models to generate accurate responses and reduces the likelihood of hallucinations while also mentioning how adding improper context leads to more hallucinations.

3 Data Sets

This project does not follow the traditional data analysis workflow. Instead, it focuses on building tools that sit on top of MI's existing data. It's meant to give users the ability to create insights for their own departments and use cases. The goal was never to run analytics on the company's webstore or pull insights from the documents they heavily rely on. Instead, it was about removing the technical barriers and slow manual processes the current framework depends on, and giving users the ability to find these insights themselves. Because of this, our **exploratory data analysis (EDA)** looks a bit different. Instead of doing a statistical deep dive into the data, we had to understand the structure, the organizational flow, and where the data actually lives so that the agents have what they need to generate **correct**, **complete**, **consistent**, and **grounded explanations** for the user. These four metrics we focused on became the pillars of our validation and our drivers for adoption, which will be discussed in more detail in Section 6.

3.1 NL2SQL Data Source

3.1.1 Source

The NL2SQL Agent is powered by Molecular Instruments' **PostgreSQL** webstore database schema, which encompasses the layout of all records pertaining to user interactions as well as order placement and fulfillment workflows. Although the

database resides in the **Google Cloud Platform (GCP)**, we generated a local SQL dump of the staging environment to protect proprietary data and streamline development, allowing team members to work without relying on the cloud layer. Documentation on how to perform a SQL data dump from GCP can be found here [Goo25d]. The dataset for the NL2SQL Agent can be found here.

3.1.2 Structure and Features

At the core of our data workflow is a database schema that is automatically parsed and represented as **JSON** via Google's open-source **Model Context Protocol (MCP) toolbox** [Goo25b]. We discuss this process in more detail in the technical and implementation section 4.2.1. At a conceptual level, this tool allows us to provide the schema data as context to the AI agents. In total, the database contains 61 tables, and each table appears as one object within the JSON array.

The information encapsulated inside the JSON array is the following:

- Relational schema consisting of entities that include information about customers, orders, order items, quotes, discounts, product tables, etc.
- Each table contains custom metadata such as column details, indexes, and foreign keys that define relationships with other tables.
- The JSON snippet on the right is a condensed example of one table in the schema that contains information produced by the MCP toolbox after translating the underlying ER diagram, seen in Figure 6.

```
[{
  "object_name": "users",
  "columns": [
    {"column_name": "id", "data_type": "uuid"},
    {"column_name": "name", "data_type": "text"},
    {"column_name": "organization", "data_type": "text"}
  ],
  "constraints": [
    {
      "constraint_columns": ["organization"],
      "constraint_definition": "CHECK ((organization <> ''::text))",
      "constraint_name": "users_organization_check",
      "constraint_type": "CHECK"
    }
  ],
  "indexes": [
    {
      "index_columns": ["id"],
      "index_method": "btree",
      "index_name": "users_pkey",
      "is_primary": true,
      "is_unique": true
    }
  ]
}]
```

*A high-level view of the ER diagram in Figure 6 is included in the Appendix 9.2. Since the full schema is quite complex, it can be explored if the reader is interested. However, our focus is on building a framework that sits on top of the database and can be applied on any PostgreSQL setup, allowing readers to reproduce the implementation on their own databases. More detailed schema notes can be found in the prompts.py file of the NL2SQL agent directory in the **Bitbucket** repository as well.*

3.1.3 Notable Dataset Nuances

We faced several challenges while preparing the data due to the high dimensionality stemming from many interconnected entities, multiple join paths, and semantic complexity. MI-specific terminology (like "HCR product lines") must be mapped to actual tables and columns and the agent must also be able to handle general variation in how users phrase their questions. Because of this, the agent must interpret intent while still grounding its outputs strictly in the schema.

3.2 FAQ RAG Data Source

3.2.1 Source

For the FAQ RAG agent, we gathered all the relevant company documents that the MI support team usually relies on when answering customer questions. These documents come in different formats, either **PDF** or **HTML**, and fall into several categories of publicly available material such as web pages, blog posts, user guides, protocols, safety data sheets, peer-reviewed research papers, and MI's own published papers. We built a simple scraper that grabs the PDFs directly and saves the HTML as plain text. The dataset for the FAQ agent can be found here.

The HTML turned out to be much messier and harder to chunk cleanly, so for this first POC we focused primarily on the PDFs. More details on this decision is explained in the technical section of the paper 4.3.1.

3.2.2 Structure and Features

The document corpus consists of **48 PDFs**, and none of them follow one single structure: research papers typically follow a conventional scientific layout with long paragraphs, sectioned narratives, and hierarchical headings, whereas protocol guides and user manuals contain shorter statements, table-heavy layouts, and more fragmented instructional text.

To facilitate experimentation, the documents were grouped into **six categories** based on content type and structural similarity. This grouping enabled us to track retrieval behavior across document classes and better understand how the agent performs when ingesting materials of different styles.

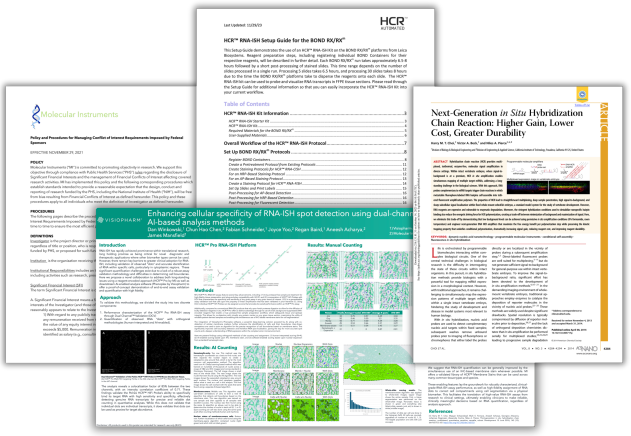
3.2.3 Notable Dataset Nuances

We needed to determine the parameters for chunking, which required an initial EDA of the documents and reference to industry standards. With a 2,000-character chunk size and a 300-character overlap, the 48 PDFs (662 total pages) generated 1,106 chunks. The summary statistics for these chunks are shown in Table 1. The reason behind these decisions and parameter choices can be found in section 4.3.3

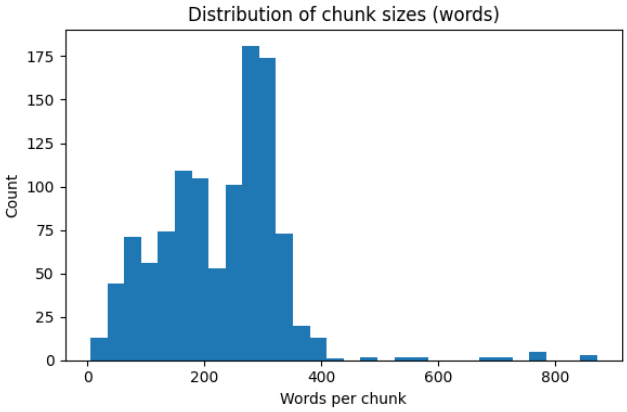
Table 1: Summary statistics for generated chunks

Metric	Min	Median	Mean	Max
Characters per chunk	48	1627.5	1454.0	2000
Words per chunk	5	244	229.1	871
Tokens per chunk (approx.)	14	426.5	427.0	944

Most chunks fall in the 1,400–1,600 character range (roughly 220–250 words). Very small or very large chunks typically originate from protocol-style PDFs or heavily formatted sections. We also approximated token counts and found that an average chunk contains about 400–430 tokens, with the full corpus totaling approximately 472,000 tokens. These estimates helped us estimate embedding costs and ensured stable retrieval.



(a) Example pages from the PDF corpus.



(b) Distribution of chunk sizes (words).

Figure 1: Visual overview of the source documents and the resulting chunk-size distribution.

4 Technical Approach / Methods

4.1 Implementation Framework

To streamline development, we used the Google ADK framework, which abstracts much of the complexity of orchestrating agent workflows. This allowed us to focus on the Agent's logic and behavior rather than building coordination infrastructure, and helped us quickly deliver a POC that fits the existing MI workflows.

Google ADK is **model-agnostic**, meaning any LLM can be used, although it is optimized for Gemini. We chose **Gemini 2.5-flash** for its free tier, more permissive rate limits compared to alternatives like OpenAI, and its large context window of 1 million tokens. The larger context window is essential for both agents since it allows our NL2SQL agent to ingest the full schema, schema notes, and example queries in a single prompt, simplifying our design and avoiding complex context-splitting strategies. For the FAQ agent, it allows for larger chunk sizes and more overlap for the retrieved chunks by the vector database.

4.2 NL2SQL Agent

4.2.1 Data Preprocessing and Setup

Before integrating our NL2SQL agent, we first needed a reliable method for dynamically loading database schema information and filtering out unnecessary schema information. These requirements stemmed from the frequent updates made to the database schema, as well as the need to minimize token consumption when passing schema context to the agent. Since operational cost is a major factor in adopting agentic workflows, ensuring efficient token usage was essential. To address these concerns, we leveraged the **Model Context Protocol (MCP) Toolbox**. MCP is an open-source standard that enables AI systems to interface with external applications and data sources. In our architecture, MCP provides a unified mechanism for connecting our agentic application to the database layer, whether running locally during development or

hosted in Google Cloud SQL. Instructions for initializing the database, importing data, and configuring a local database connection can be found in the project repository README. To learn how to startup the **MCP toolbox Docker container** you can reference the MCP toolbox docs here.

Once the database and toolbox are running and connections to them are established, we invoke an MCP tool that enumerates all tables in the database schema, returning them in the structured format described in 3.1.2. From this full schema, we prune the tables that are never queried and are irrelevant to analytics at Molecular Instruments. After **schema pruning**, we removed extraneous whitespace from the JSON representation to further reduce token usage. At this point, the cleaned and compact schema is ready for consumption by the NL2SQL agent. Following preprocessing, the schema shrank from approximately **90,000 tokens** to **70,000 tokens**. A cost breakdown of token usage can be found here.

4.2.2 Architecture Overview

The architecture shown in Figure 2, represents the implementation used to build our NL2SQL agent. In the typical workflow, an **MI administrator** submits a question related to the MI database. Using schema notes, business terminology, example queries, and the schema loaded dynamically via the MCP toolbox, the NL2SQL agent then generates a candidate SQL query intended to answer the user's question.

Once a query is produced, it must be validated to ensure that it is executable, which prevents issues such as syntax errors, nonexistent tables, or fabricated column names. This validation step is handled by the Execute Query component. If the query executes successfully, the system returns the SQL code along with a formatted explanation. If execution fails, the system enters a refinement loop: the agent attempts to correct the query, after which the updated query is executed again to verify that the refinements were successful. To avoid infinite refinement cycles, the loop is limited to a maximum of two iterations. If the agent is still unable to produce a syntactically and schematically valid query after two attempts, it is considered unlikely to reach a correct solution for the prompt. We chose two iterations because further rounds introduce significant computational cost and slow the system's latency without meaningfully improving success rates.

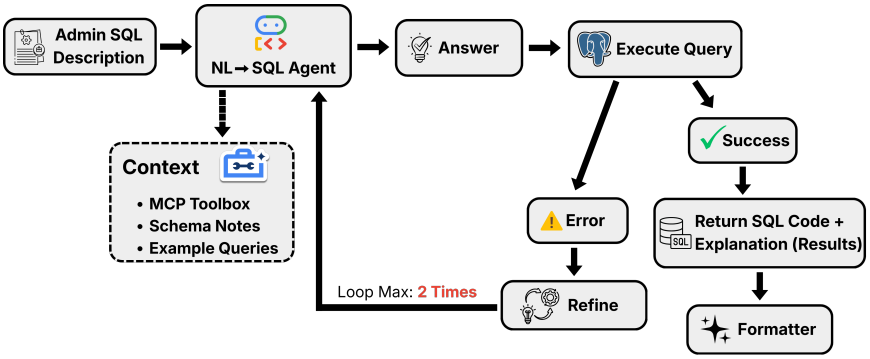


Figure 2: NL2SQL Agent Architecture

4.2.3 Design Considerations

Several design considerations shaped how the NL2SQL Agent was built. Many of these components were not part of our initial design but became necessary once we saw how the system behaved in practice. Each addition influenced the overall architecture and improved reliability, grounding, and user experience.

- **Loop Agent:** Our initial design returned the first SQL query that executed successfully, but this proved unreliable since even small errors could slip through and refinements were never re-validated. To address this, a Loop Agent that iteratively refines and re-executes each query before returning it to the user was added. This became a core part of the architecture, ensuring correctness and providing a safety net for ambiguous or complex questions.
- **Context Caching:** Because the schema is large and LLM costs scale with token count, we needed a way to avoid repeatedly sending unchanged prompt components such as schema notes. Context caching [Goo24a] solves this by storing designated sections of a prompt. We can configure both the token threshold that triggers caching and the cache duration. Given the size of our schema, we enable caching only when the prompt exceeds **60,000 tokens** and retain the **cache for 6 hours (21,600 seconds)**. Caching the schema notes for an extended period is acceptable since the schema doesn't frequently get updated.
- **Temperature tuning:** We intentionally used a **temperature of 0** to keep the agent as deterministic as possible. Since SQL generation benefits from precision rather than creativity, lowering the temperature minimizes variability across runs and reduces hallucinations.
- **Structured JSON Passing:** Strict `output_key` fields and schema validation were enforced to ensure agents could pass structured JSON reliably across the pipeline. A Formatting Agent was then added to standardize the final result into clean SQL plus a short explanation, giving the UI (and, in turn, the user) a predictable, well-structured output even when earlier steps are probabilistic.

Together, these choices helped balance determinism and flexibility. The agent remains grounded through schema-aware context, predictable through formatting and low temperature, and resilient through the loop-based refinement process, all while minimizing cost and reducing hallucinations.

4.3 FAQ RAG Agent

4.3.1 Data Preprocessing

Because the corpus contains both long-form scientific papers and highly structured user guides, preprocessing was required to normalize document text, extract metadata, and prepare the content for chunking and embedding. Metadata stored alongside each chunk includes document name, source category, and format type, allowing downstream filtering and evaluation. **Chunking** is the process of splitting long documents into smaller text segments that language models can embed, index, and retrieve efficiently. To help us visualize chunking behavior across the layouts, the tool: Visualize Chunking was used. Each PDF is represented as a collection of clean, well-structured chunk embeddings that the agent can pull from when generating answers.

4.3.2 Architecture Overview

The FAQ RAG agent follows a standard retrieval-augmented generation pipeline. When a user submits a question, the system first retrieves the most relevant segments from the document corpus and then generates an answer strictly grounded in those retrieved chunks.

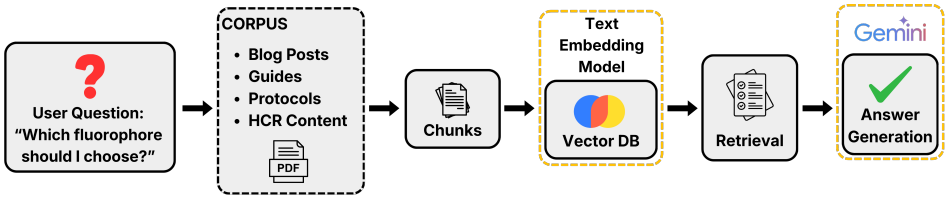


Figure 3: FAQ RAG Agent Architecture

The pipeline consists of the following steps:

1. **Chunking & Indexing** All documents are split into smaller text units ("chunks") and stored in a vector database. We used **ChromaDB** as the vector store and Gemini embeddings to encode text into high-dimensional vectors. The specific Gemini text embeddings model we use in our vector database is: **text-embedding-004**.
2. **Query Encoding & Retrieval** At query time, the user's question is converted into a vector using the same embedding model. Vector similarity, computed using cosine similarity, is then used to identify the closest document chunks within the corpus. These **top 5** ranked chunks represent the context passed to the LLM.
3. **LLM Answer Generation** The LLM generates an answer based solely on the retrieved context, following strict prompting guidelines designed to minimize hallucinations and ensure factual grounding.

This architecture maintains a clear separation between retrieval and reasoning, ensuring traceability and interpretability of each answer.

4.3.3 Design Considerations

Because our corpus includes documents with dramatically different structures and lengths, the agent must embed both short and long textual segments effectively. To accommodate this, we opted for a relatively large chunk size of 2,000 characters with an overlap of 300 characters. Chunking was implemented using a naive baseline: the **RecursiveCharacterTextSplitter**, which recursively searches for natural breakpoints (paragraphs, lines, spaces) before falling back to raw character boundaries [Chr24] [Dat24].

```
splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=300)
```

This approach was chosen because it is simple, robust across diverse document types, and computationally efficient. However, it does not capture document semantics or structural cues. Refer to *Future Works* for different chunking approaches 8.3. As part of **hallucination mitigation**, we maintained a low **temperature range (0.0–0.2)**, which reduces speculative reasoning while still allowing the model to paraphrase and synthesize information across retrieved chunks. To enforce grounding, we implemented a strict prompt:

```
instruction=(
    "Always base your answers ONLY on the retrieved context. "
    "You MUST NOT invent information or make probabilistic assumptions. "
),
prompt = ChatPromptTemplate.from_messages([
    ("system",
     "You are an expert assistant for Molecular Instruments HCR™ workflows.\n\n"
     "Use ONLY the provided context to answer each question,\n"
     "but answer in your own words.\n"
     "Summarize and synthesize information from multiple parts of the context if needed.\n"
     "CRITICAL RULES:\n"
     "- Do NOT infer new facts from indirect evidence or implications.\n"
     "- Avoid speculative language such as 'likely', 'probably', or 'it suggests that'.\n"
     "- If the documents do not state the answer explicitly, .\n"
     "respond that the information is not specified.\n"
     "If you cannot find sufficient information in the context, say you don't know."
    ),
])
```

This combination of controlled temperature and strongly constrained prompting helped maintain factual accuracy and reduce hallucinations despite the heterogeneous corpus.

5 Software and Codebase

5.1 Directory Structure

Our codebase is organized into two agent subdirectories, each containing the necessary modules for implementation, validation, and data storage (condensed view, more detail in Bitbucket repository):

```
agents/
├── .env                            # Shared environment variables (API keys, model name)
├── tools.yaml                      # MCP toolbox configuration
├── requirements.txt
├── faq_agent/
|   ├── __init__.py
|   ├── faq_agent.py                # Root FAQ agent definition
|   ├── rag_langchain.py            # RAG pipeline: loading, chunking, retrieval
|   ├── validate_faq_agent.py       # Evaluation + test harness
|   └── webpages/                   # Raw webpage/PDF corpus (48 PDFs)
└── nl2sql_agent/
    ├── __init__.py
    ├── agent.py                    # NL2SQL multi-agent pipeline (writer → executor → validator)
    ├── prompts.py                  # Instruction templates
    ├── nl2sql_toolbox_tools.py     # MCP schema tools (list_tables, get_enum_types)
    ├── generate_agent_output.py    # Batch generation utility
    ├── test_queries.csv            # Official 22 evaluation queries
    └── test_queries_template.csv   # Blank template for creating new test cases
```

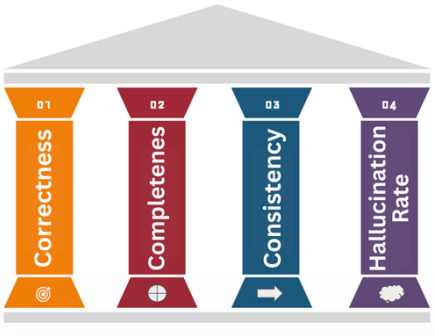
5.2 Core Libraries and Tools

- **Google Agent Development Kit (ADK):** Orchestration framework for coordinating multi-agent workflows (includes a built-in prototyping UI ADK Web). A clean, formal example adhering to best practices from the Google ADK documentation (including multiple sample agents) is available here.
- **Model Context Protocol (MCP) Toolbox:** Open-source standard enabling dynamic loading of database schema information. Guidance on how to write a YAML file for the MCP toolbox can be found in the official documentation here.
- **psycopg2:** PostgreSQL adapter used for executing SQL queries and validating results [The25].
- **LangChain:** Framework for orchestrating the RAG pipeline, including document loading, chunking, embedding storage, and retrieval. [Lan25a]
- **PyPDFLoader:** Loader for extracting and parsing PDF documents with page-level metadata [Lan25b].
- **RecursiveCharacterTextSplitter:** Splits documents into overlapping chunks (2,000 characters with 300-character overlap) using recursive breakpoint detection to preserve semantic boundaries [Lan25c].
- **Chroma (ChromaDB):** Vector database used for storing and retrieving document embeddings [Chr25].
- **GoogleGenerativeAIEmbeddings:** Embedding model (`text-embedding-004`) that converts text into high-dimensional vectors for semantic similarity search [Goo25a].
- **Gemini 2.5-flash:** Primary LLM selected for its free tier and 1M-token context window [Goo25c].
- **Pydantic:** Enforces structured JSON outputs and type safety across the agent pipeline [Pyd25].
- **MiniLM (all-MiniLM-L6-v2):** Lightweight transformer model used during FAQ evaluation to compute semantic similarity via cosine distance between sentence embeddings [Trand].
- **python-dotenv:** Utility for managing environment variables, API keys, and configuration files [Kum25].

6 Evaluation

6.1 Drivers of Adoption

To align our evaluation with MI's requirements, we assess both agents across four pillars: correctness, completeness, consistency, and hallucination rate. These metrics were selected because they reflect what stakeholders care about most and because each can be scored in a clear, discrete way that supports easy interpretation of results.



- **Correctness.** We compare each output to an expert-validated reference. For the NL2SQL agent, we use a 1–4 score: **4** = fully correct; **3** = minor edits; **2** = partially on track but missing important logic; **1** = wrong or unusable. For the FAQ RAG, the score is **1** if the answer accurately reflects the source material and **0** otherwise.
- **Completeness.** We ask whether the output fully answers the question. NL2SQL queries must include all required fields and conditions, while FAQ RAG answers must cover all key aspects of the question with supporting citations, making citation accuracy a core signal of completeness.
- **Consistency.** We re-ask each question multiple times to assess stability. For NL2SQL, we check for logically equivalent queries and similar results; for FAQ RAG, we look for semantically similar answers that rely on comparable context and citations.
- **Hallucination rate.** We track how often the model introduces unsupported information. NL2SQL hallucinations include nonexistent tables, columns, or relationships; FAQ RAG hallucinations occur when the agent states facts not grounded in the corpus or answers out-of-scope questions without acknowledging uncertainty.

6.2 Validation of NL2SQL Agent

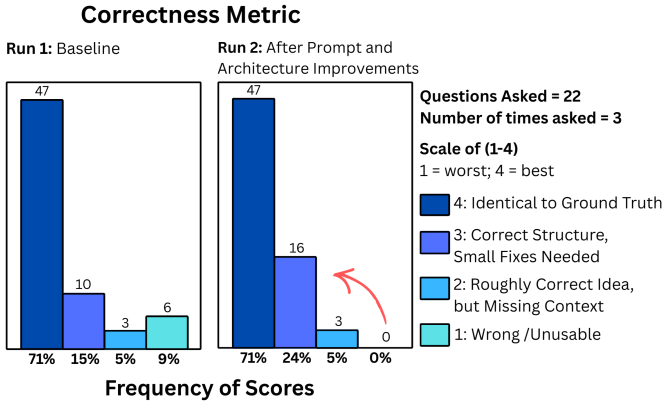
6.2.1 Strategy

To evaluate the reliability and robustness of our NL2SQL agent, we developed a structured validation suite designed to test correctness, consistency, and completeness of the generated SQL queries. The current suite consists of **22 test questions**, each derived from real query requests previously submitted by users. This ensures that the evaluation reflects practical use cases rather than synthetic or overly simplified examples. For each question, we run the agent across three independent sessions, producing **66 total agent outputs**. Repeating each query multiple times allows us to measure consistency, i.e., whether the agent generates stable SQL outputs under identical input conditions. This is particularly important for assessing nondeterministic model behavior.

To assess correctness, every generated SQL query is compared against a manually constructed ground-truth query that has been validated by domain experts. This comparison ensures that the agent's output not only executes successfully but also retrieves the intended data with the correct logic, filters, joins, and aggregations. Finally, we evaluate completeness by checking whether each SQL query fully addresses all aspects of the natural-language request. A query may be syntactically correct yet incomplete if it omits conditions, fails to incorporate required attributes, or only partially fulfills the data retrieval intent. Completeness scoring ensures the agent captures the full scope of the user's request. This validation framework will continue to expand as additional questions and more rigorous test cases are incorporated.

6.2.2 Key Findings

The metrics for the NL2SQL Validation are summarized below:



Completeness, Consistency & Hallucination Rate Metrics

Completeness (across outputs)	**57 Complete 9 Incomplete**
Consistency (across questions)	**17 Fully consistent 3 Partially consistent 2 Inconsistent**
Hallucination Rate	**0 Queries contained hallucinations**

Figure 4: Validation Process and Metrics for FAQ RAG Agent

Across all outputs, the agent achieved an **average correctness score of approximately 3**, which is computed as the sum of correctness scores divided by the total number of generated queries. While the initial overall performance is strong, particularly the high proportion of fully correct queries, the presence of several low-scoring outputs suggests that the agent still struggles with specific cases requiring more nuanced contextual understanding. These errors highlight opportunities for improvement such as prompt refinement, schema grounding, and handling of complex or ambiguous query requests. We were able to improve the model and add missing context and in turn eliminated the queries that got a score of 1. However, our test suite remains subject to expansion, which was not completed within the current time frame. Stakeholders expressed satisfaction with the current performance; however, we expect that continued iteration and targeted adjustments can further improve correctness and reliability in future versions of the NL2SQL agent.

6.3 Validation of FAQ RAG Agent

6.4 Strategy

Across ten iterative versions of the agent, we adjusted prompting strategies, chunking parameters, corpus size, question composition, and temperature settings to evaluate how each component contributes to overall behavior. For each version, we executed five independent agent sessions to assess consistency, defined as the stability of answers across repeated runs. Each session included 10 randomly sampled questions from a curated validation set, consisting of:

- **Factual questions**: 15 questions per document whose answers are explicitly present in the corpus. For version 10 of the agent, the validation set included 170 questions across 11 documents, evenly distributed by document type.
- **Out-of-scope questions**: 5 questions per run, for which the correct answer is "I don't know." These measure Hallucination Rate.

Each generated answer was scored along three dimensions:

- **Correctness**: measured via MiniLM [Trand] which is a compact transformer model that produces sentence embeddings. We embed both the expected answer and the agent's answer with MiniLM and use cosine similarity between the two embeddings as our correctness score.
- **Citation Accuracy**: verifying whether the retrieved chunk(s) mapped to the correct PDF and page.
- **Hallucination Score**: proportion of out-of-scope questions answered correctly with "I don't know".

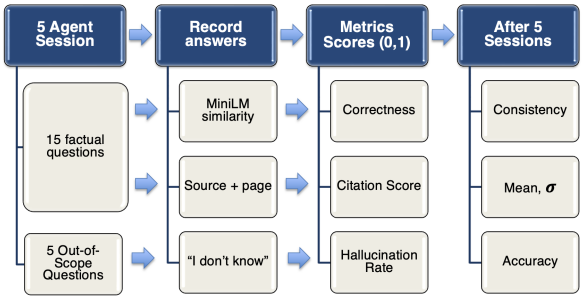


Figure 5: Validation Process and Metrics

6.5 Progress Across Iterative Versions

Early versions (V1–V2) achieved high accuracy when evaluated on a single long-form research paper. However, introducing a second document immediately revealed generalization weaknesses: correctness dropped from **1.00 to 0.21**, even though hallucination and citation scores remained high. This demonstrated that the initial prompt and chunking strategy were overfit to uniform, paragraph-based text and could not handle heterogeneous structures such as user guides or safety data sheets.

Middle versions (V3–V6) introduced improvements to prompt flexibility, chunk size, and question sampling. Slightly increasing the temperature to **0.2** improved contextual reasoning and synthesis, while raising chunk size from **900 to 2,000 tokens** preserved semantic coherence in long documents. Accuracy improved from **0.59 to 0.85**, although hallucinations became more frequent when the agent inferred conclusions from indirect cues. This reflects the well-known tradeoff: larger context windows aid retrieval but increase speculative risk.

Version 7 exposed weaknesses in the validation set itself. Several ambiguous questions were incorrectly classified as control questions, causing the agent to appear to hallucinate. After refining the dataset and increasing the proportion of control questions to **3 out of 10**, hallucination detection improved substantially.

Versions 8–10 focused on eliminating speculative behavior. A refined prompt explicitly prohibited inference, implication-based reasoning, and probabilistic language. The validation question set was further cleaned to remove ambiguous items. Scaling the corpus to include five long-paragraph and five short-paragraph documents tested the agent under realistic multi-document workloads. At a temperature of 0.1, Version 9 achieved the highest correctness **(0.94)** but suffered from reduced citation precision due to broader retrieval. Version 10 delivered the best-balanced performance **(0.86 correctness, 0.79 hallucination rate, 0.80 citation score)**.

6.6 Key Findings

Overall, the validation stage demonstrates that:

1. **Prompt constraints** reduce hallucinations but must be balanced with conciseness requirements to maintain high semantic similarity.

Table 2: FAQ RAG Performance Snapshot

Version	Correctness	Hallucination Score	Citation Score	Temp
V1	1.00	1.00	1.00	0
V2	0.21	1.00	1.00	0
V6	0.85	0.78	1.00	0.2
V9 (best correctness)	**0.94**	0.57	0.51	0.1
V10 (best balanced)	**0.86**	0.79	0.80	0.1

2. **Chunk sizing** has different optimal configurations for long-form research papers versus short instructional documents.
3. **Validation quality and question design** substantially influence measured performance.
4. **Scaling to heterogeneous document types** remains the major challenge for RAG systems in specialized scientific domains.

The strongest performance emerged under **low temperature (0.1), non-speculative prompts, chunk sizes tuned to document structure, and a diverse, well-curated mix of factual and control questions**. These insights inform the next stage of development, including multi-document retrieval optimization at scale, prompt adjustments for concise, grounded answers, and expanded evaluation metrics for production readiness.

Tradeoffs Driving Accuracy and Hallucinations During evaluation, we observed a clear tradeoff between temperature settings, chunk size, prompt strictness, and agent behavior. This pattern is illustrated in Figure 9, which shows how hallucination rates shifted as we adjusted temperature across agent versions. At zero temperature, the agent was highly deterministic and avoided speculative patterns but occasionally produced rigid responses, especially when the retrieved context required synthesis.

Increasing the temperature even slightly improved contextual reasoning, while increasing chunk size helps preserve semantic coherence, leading to higher correctness scores. However, larger chunks also exposed the agent to more contextual noise, making speculative errors more likely. For example, the agent was inferring unsupported claims about international shipping, illustrating a well-documented RAG pattern: **larger context windows aid retrieval but amplify inference risk** when prompt rules are insufficiently strict.

Our optimal configuration combined **low temperature (0.1–0.2)** with **strong anti-speculation prompts** and **chunk sizes tuned to document type**. This setup enabled the agent to synthesize information accurately without a substantial rise in hallucinations [But24, Fli25].

7 Team Member Participation

Name	Primary Contributions	NL2SQL	FAQ	Slides/Report
Mohamad Chaalan	Led NL2SQL agent & validation logic; performed initial setup for both agents, including data gathering, Postgres config & schema preprocessing	25%	5%	20%
Angela Lombard	Contributed to NL2SQL design; built the UI and created the visualizations	25%	0%	20%
Anastassiya Iskaliyeva	Led the FAQ agent; chunking strategy, retrieval pipeline, prompting, and extensive testing	0%	50%	15%
Edom Eshete	NL2SQL testing, schema preparation, debugging, and refinement assistance	25%	0%	15%
Kenny Gao	FAQ agent exploration and testing, vector store experiments, and retrieval experimentation	0%	35%	15%
Richard Wicaksono	Support on both agents with a focus on MCP toolbox integration; EDA and data exploration	25%	10%	15%

Our team collaborated closely throughout the quarter, with each member contributing to different parts of the project in ways that collectively shaped the final system. Early development of the NL2SQL agent was initiated by Edom, and Mohamad extended this work by setting up the database environment, preparing the schema, and implementing the core validation pipeline. Richard contributed to the NL2SQL agent by integrating the MCP toolbox for dynamic schema access, and Angela added the loop-agent refinement that helped improve the workflow. For the FAQ RAG agent, both Anastassiya and Kenny worked on the initial structure and experimentation, and together they played key roles in refining the architecture, improving retrieval behavior, and strengthening the prompting and testing process. We all pushed code to our individual Bitbucket branches, with Mohamad managing the final merges since he is most familiar with the company's Git workflow. Throughout the project, every team member consistently participated in meetings, debugging, writing, and collaborative work sessions, making it a genuinely shared and positive team effort.

8 Discussion and Conclusion

8.1 Challenges

NL2SQL – A difficulty we faced was balancing determinism with flexibility. The NL2SQL agent requires rigidity and precision rather than creativity, especially when generating consistent queries and maintaining semantic understanding. When a user's input is rigid and specific, the agent can mirror that rigidity; however, when the input is vague or ambiguous, the agent must infer missing intent, which introduces risk and uncertainty. To mitigate this, we designed the NL2SQL agent to be more strict and to require users to ask clear, well-specified questions.
FAQ Agent – The corpus contains overlapping or duplicated content. As more documents were added, this redundancy produced ambiguous retrieval signals, sometimes returning multiple equally valid chunks for the same question. The presence of multiple acceptable answer sources complicated citation validation. Even when the agent produced a correct answer, the evaluation script sometimes marked it as incorrect because the citation matched a duplicate chunk rather than the expected one. This challenge is well-known in scaled RAG systems, and several mitigation strategies are mentioned in the Section 8.3, which may help future versions.

8.2 Limitations

NL2SQL – **(1)** Further validation is required to evaluate performance across a broader and more diverse set of queries. In particular, we need to ensure consistent accuracy across all tables in the database and achieve sufficient coverage of the full schema, rather than optimizing for a limited subset of frequently used tables. **(2)** Due to time constraints, it was not feasible to map all edge cases in the underlying business logic. The current implementation prioritizes the most critical and frequently accessed workflows. Additional work is needed to extend business-rule mappings to less common scenarios and ensure comprehensive logical coverage.
FAQ Agent – Beyond the previously discussed challenge of **overlapping content across documents**, which complicates retrieval and citation accuracy, a major limitation of the validation stage is that **large-scale testing is difficult to perform under free or quota-limited compute resources**. Each evaluation run requires embedding, retrieval, and inference across multiple PDFs, which becomes slow, resource-intensive, and prone to rate-limit interruptions as the corpus expands. As MI evaluates next steps for deployment, the ability to run high-volume, reproducible validation experiments will be essential for ensuring reliability, safety, and scalability. These limitations highlight the need for more structured preprocessing, scalable retrieval infrastructure, and more controlled generation strategies that explicitly balance temperature, chunk size, and prompt design as MI expands the RAG agent beyond the POC stage.

8.3 Future Work

If we had more time, our next steps would focus on deepening reliability and expanding each system's capabilities. For **NL2SQL Agent**, this would include more advanced schema linking, improved query decomposition for multi-step questions, and larger validation suites sourced from real user queries. For the **FAQ RAG Agent**, we would explore several options to improve performance. **(1)** Semantic or structure-aware chunking, multi-document retrieval optimization, and stronger hallucination guards such as confidence estimation or answer-ability detection [PCXW25]. These methods could improve retrieval precision, though such techniques often introduce scalability challenges. **(2)** Deduplication of repeated passages prior to embedding [NVI24] **(3)** awareness of retrieval degradation when retrieving many documents [LMS+25] **(4)** Additional tooling such as document ingestion pipelines, or hybrid RAG/database queries could further strengthen reliability. Both agents would also benefit from long-term monitoring pipelines to track drift, user feedback, and failure patterns over time. MI could also integrate both agents into a unified internal assistant, capable of switching between SQL generation and document-grounded answers seamlessly.

8.4 Conclusion

Building this agentic AI framework gave us a much deeper understanding of what it means to design practical, reliable AI tools for real organizational workflows. Each agent revealed layers of complexity that required careful engineering, thoughtful design choices, and frequent iteration. We learned that Agentic systems are not just about model quality, they depend heavily on data preprocessing, schema clarity, chunking strategies, validation loops, grounding rules, and temperature control. Overall, developing these agents gave us not only functional prototypes but also a deeper understanding of the tradeoffs, engineering decisions, and validation strategies required to build these AI systems. While our methods may not be perfect yet, both agents have already demonstrated potential for real value to our sponsors, and their positive feedback reinforces the potential of continuing this work.

9 Appendix

9.1 references/bibliography

References

[But24] Danny Butvinik. Mathematical insights into temperature scaling and hallucination in large language models, 2024. Accessed: 2025-12-08.

[Chr24] Chroma. Evaluating chunking strategies for retrieval-augmented generation. https://research.trychroma.com/evaluating-chunking, 2024. Accessed: 2025-12-08.

[Chr25] Chroma. Chromadb documentation. https://docs.trychroma.com/, 2025. Online documentation; accessed 2025-12-10.

[Dat24] Databricks. A compact guide to retrieval-augmented generation (rag). https://www.databricks.com/sites/default/files/2024-05/2024-05-EB-A_Compact_GuideTo_RAG.pdf, 2024. Accessed: 2025-12-08.

[Fli25] Justin Flitter. Why ai models hallucinate and how to prevent it, 2025. Accessed: 2025-12-08.

[Goo24a] Google Inc. Adk context caching. https://google.github.io/adk-docs/context/caching/, 2024. Agent Developer Kit (ADK) Documentation; accessed 2025-12-10.

[Goo24b] Google Inc. Agent developer kit (adk) documentation. https://google.github.io/adk-docs/agents/, 2024. Online documentation.

[Goo25a] Google AI. text-embedding-004. https://ai.google.dev/api/embeddings, 2025. Google Generative AI Embeddings; accessed 2025-12-10.

[Goo25b] Google APIs. Genai toolbox — getting started (introduction). https://googleapis.github.io/genai-toolbox/getting-started/introduction/, 2025. Online documentation; accessed 2025-12-07.

[Goo25c] Google DeepMind. Gemini 2.5 flash. https://ai.google.dev/models/gemini, 2025. Large language model; accessed 2025-12-10.

[Goo25d] Google Inc. Export and import using sql dump files. https://docs.cloud.google.com/sql/docs/postgres/import-export/import-export-sql, 2025. Online documentation; accessed 2025-12-08.

[Jac25] Per Jacobsson. Techniques for improving text-to-sql. https://cloud.google.com/blog/products/databases/techniques-for-improving-text-to-sql, May 2025. Google Cloud Blog; Principal Software Engineer.

[Kum25] Saurabh Kumar. python-dotenv. https://github.com/theskumar/python-dotenv, 2025. Environment variable management for Python; accessed 2025-12-10.

[Lan25a] LangChain Developers. Langchain documentation. https://docs.langchain.com/, 2025. Online documentation; accessed 2025-12-10.

[Lan25b] LangChain Developers. Pypdfloader. https://docs.langchain.com/oss/python/integrations/document_loaders/pypdfloader, 2025. LangChain documentation; accessed 2025-12-10.

[Lan25c] LangChain Developers. Text splitters. https://docs.langchain.com/oss/python/integrations/splitters/index#text-splitters, 2025. LangChain documentation; accessed 2025-12-10.

[LMS+25] Shahar Levy, Nir Mazor, Lihi Shalmon, Michael Hassid, and Gabriel Stanovsky. More documents, same length: Isolating the challenge of multiple documents in rag, 2025. Accessed: 2025-12-08.

[NVI24] NVIDIA. Rag 101: Retrieval-augmented generation — questions answered. https://developer.nvidia.com/blog/rag-101-retrieval-augmented-generation-questions-answered/, 2024. Accessed: 2025-12-08.

[PCXW25] Xiangyu Peng, Prafulla Kumar Choubey, Caiming Xiong, and Chien-Sheng Wu. Unanswerability evaluation for retrieval augmented generation. https://arxiv.org/abs/2412.12300, 2025. arXiv preprint arXiv:2412.12300; accessed 2025-12-10.

[Pyd25] Pydantic Developers. Pydantic documentation. https://docs.pydantic.dev/, 2025. Online documentation; accessed 2025-12-10.

[RJ25] Cyrus Rashtchian and Da-Cheng Juan. Deeper insights into retrieval augmented generation: The role of sufficient context. https://research.google/blog/deeper-insights-into-retrieval-augmented-generation-the-role-of-sufficient-context/, May 2025. Google Research Blog; accessed 2025-12-07.

[The25] The Psycopg Project. psycopg2 — postgresql database adapter for python. https://www.psycopg.org/docs/, 2025. Online documentation; accessed 2025-12-10.

[Trand] Sentence Transformers. all-minilm-l6-v2. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, n.d. Pretrained model for sentence embeddings.

[ZLCL24] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. Large language model enhanced text-to-sql generation: A survey. https://arxiv.org/html/2410.06011v1, October 2024. arXiv preprint arXiv:2410.06011; License: CC BY-NC-SA 4.0.

9.2 Additional Figures or Analyses

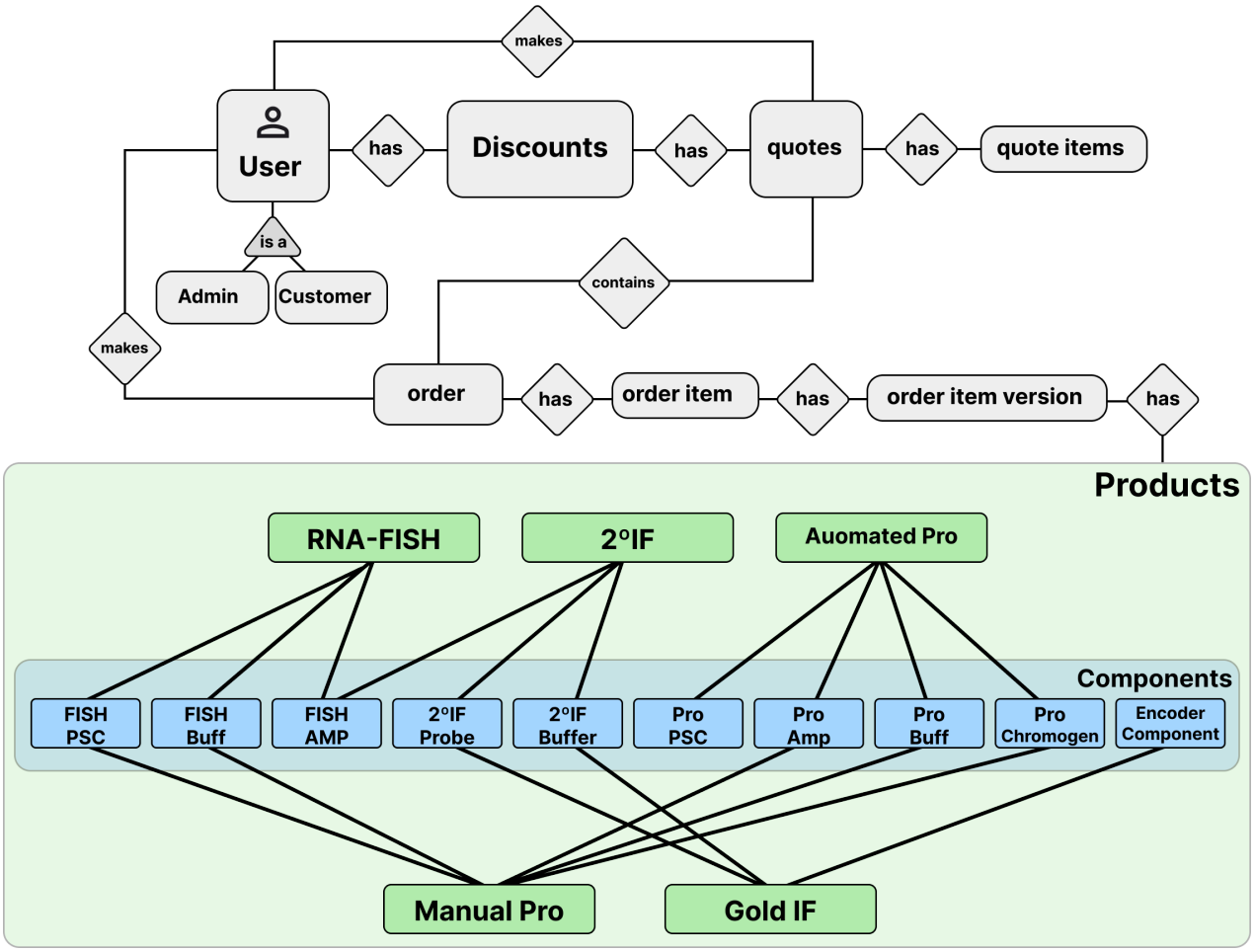


Figure 6: Webstore Database ER Diagram

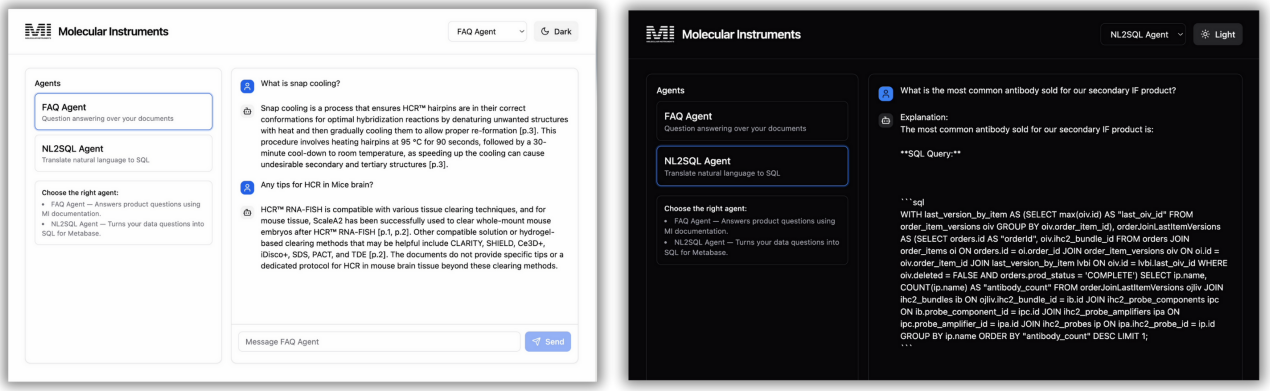


Figure 7: POC UI for Both Agents with Dark & Light Mode Feature

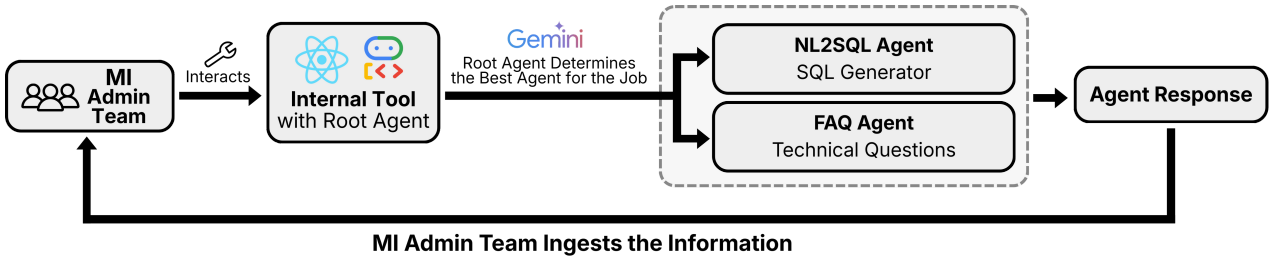


Figure 8: Agent and User Interaction - Flow of Data

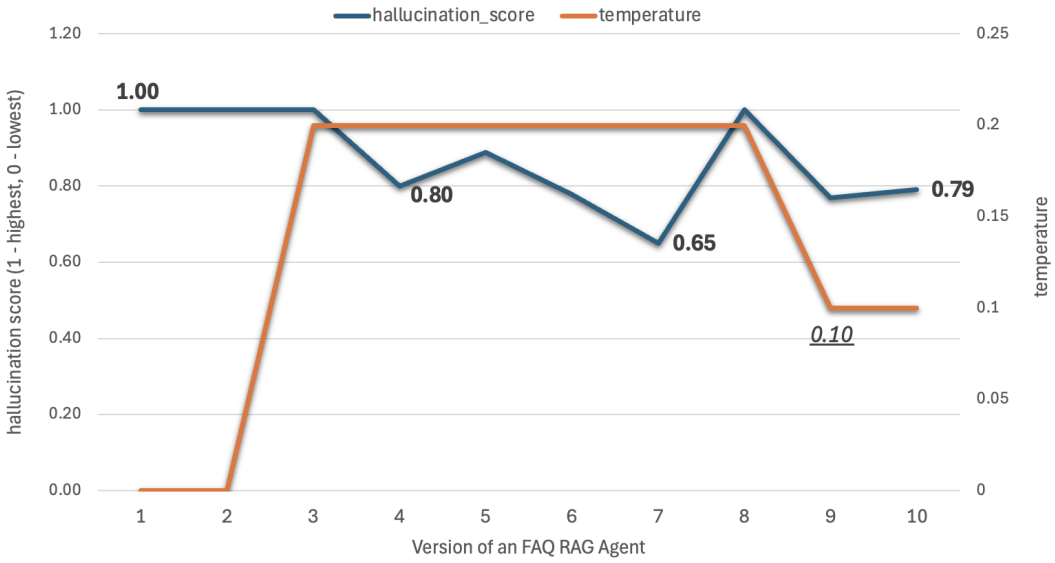


Figure 9: Hallucination & Temperature Trade off